MP 1 – Basic OCaml and Recursion

CS 440 – Fall 2006

Revision 1.0

Assigned September 7, 2006
Due September 15, 2006
Extension September 17, 2006

1 Objectives and Background

The purpose of this MP is to help the student master

1. the syntax of OCaml
   - function and variable definitions
   - lists and list operations
   - recursive definitions
2. recursive functions
   - linearity
   - integer
   - list-generating
   - mapping
   - folding
   - accumulating

The other purpose of this MP is to provide a framework to study for the exam. Several of the questions on the exam will appear shockingly similar to the problems on this MP. Your goal is to be able to solve any of these problems with pen and paper in less than 3 minutes. We recommend that you solve the problems several times to reach that proficiency.

2 Problems

2.1 Simple Functions

1. Write a function `inc` which increments its input.

```
# let inc x = ...
val inc : int -> int = <fun>
# inc 3;;
- : int = 4
```

2. Write a function `double` which doubles its input.

```
# let double x = ...
val double : int -> int = <fun>
# double 4;;
- : int = 8
```

3. Write a function `negate` which negates its input.

```
# let negate x = ...
val negate : int -> int = <fun>
# negate 34;;
- : int = -34
# negate -32;;
negate -32;;
This expression has type int -> int but is here used with type int
# negate (-23);;
- : int = 23
```

2.2 Two Argument Functions

4. Write a function `plus` which takes two arguments and returns their sum.

```
# let plus a b = ...
val plus : int -> int -> int = <fun>
# plus 2 3;;
- : int = 5
```

5. Write a function `times` which takes two arguments and returns their product.

```
# let times a b = ...
val times : int -> int -> int = <fun>
# times 3 53;;
- : int = 159
```

2.3 Tupled Form

6. Write a function `pplus` which takes a tuple of integers and returns their sum.

```
# let pplus (a,b) = ...
val pplus : int * int -> int = <fun>
# pplus (4,5);;
- : int = 9
```

7. Write a function `ptimes` which takes a tuple of integers and returns their product.

```
# let ptimes (a,b) = ...
val ptimes : int * int -> int = <fun>
# ptimes (6,9);;
- : int = 54
```

2.4 Pairs

8. Write a function `makepair` which takes two arguments and returns them in a tuple.

```
# let makepair x y = ...
val makepair : 'a -> 'b -> 'a * 'b = <fun>
# makepair 3 5.4;;
- : int * float = (3, 5.4)
# makepair "Hi" 9;;
- : string * int = ("Hi", 9)
```

9. Write a function `pi1` which takes a tuple and returns the first part.

```
# let pi1 (a,b) = ...
val pi1 : 'a * 'b -> 'a = <fun>
# pi1 ("Hi","There");;
- : string = "Hi"
# pi1 (makepair 5 6);;
- : int = 5
```

10. Write a function `pi2` which takes a tuple and returns the second part.

```
# let pi2 (a,b) = ...
val pi2 : 'a * 'b -> 'b = <fun>
# pi2 ("Hi","There");;
- : string = "There"
# pi2 (makepair 5 6);;
- : int = 6
```

2.5 Integer Recursion

11. Write a function `fact` which returns the factorial of its input.

```
# let rec fact n = ...
val fact : int -> int = <fun>
# fact 6;;
- : int = 720
```

12. Write a function `pow` which takes two arguments n and x and returns x^n. Assume that $n \geq 0$. Be careful to get the types right!

```
# let rec pow n x = ...
val pow : int -> float -> float = <fun>
# pow 2 2.0;;
- : float = 4.
# pow 2 2.5;;
- : float = 6.25
# pow 9 0.98;;
- : float = 0.833747762130149672
```

2.6 Lists

13. Write a function `isEmpty` that takes a list and returns whether or not it is empty.

```
# let isEmpty x = ...
val isEmpty : 'a list -> bool = <fun>
# isEmpty [];;
- : bool = true
# isEmpty [3;4];;
- : bool = false
```

14. Write a function `getFirst` that return the first element of a non-empty list. You do not have to check for an empty list.

```
# let getFirst x = ...
val getFirst : 'a list -> 'a = <fun>
# getFirst [3;4];;
- : int = 3
```

15. Write a function `size` that return the size of a list.

```
# let size xx = ...
val size : 'a list -> int = <fun>
# size [];;
- : int = 0
# size [2;6;3];;
- : int = 3
```

16. Write a function `incList` that takes a list $[x_0; x_1; \cdots; x_n]$ and returns the list $[x_0 + 1; x_1 + 1; \cdots; x_n + 1]$.

```
# let rec incList xx = ...
val incList : int list -> int list = <fun>
# incList [];;
- : int list = []
# incList [3;6;4];;
- : int list = [4; 7; 5]
```

17. Write a function `doubleList` that takes a list $[x_0; x_1; \cdots; x_n]$ and returns the list $[x_0 * 2; x_1 * 2; \cdots; x_n * 2]$.

```
# let rec doubleList xx = ...
val doubleList : int list -> int list = <fun>
# doubleList [];;
- : int list = []
# doubleList [2;3;2];;
- : int list = [4; 6; 4]
```

18. Write a function `sumList` that takes a list $[x_0; x_1; \cdots; x_n]$ and returns $\Sigma_{i=0}^{n} x_i$.

```
# let rec sumList xx = ...
val sumList : int list -> int = <fun>
# sumList [];;
- : int = 0
# sumList [1;2;3;4;5];;
- : int = 15
```

19. Write a function `productList` that takes a list $[x_0; x_1; \cdots; x_n]$ and returns $\Pi_{i=0}^{n} x_i$.

```
# let rec productList xx = ...
val productList : int list -> int = <fun>
# productList [1;2;3;4;5];;
- : int = 120
# productList [];;
- : int = 1
```

20. Write a function `zipLists` which takes two lists $[x_0; x_1; \cdots; x_n]$ and $[y_0; y_1; \cdots; y_n]$ and return the list of pairs $[x_0, y_0; x_1, y_1; \cdots x_n, y_n]$. If one list is longer than the other, truncate it.

```
# let rec zipLists xx yy = ...
val zipLists : 'a list -> 'b list -> ('a * 'b) list = <fun>
# zipLists [2;3;4] ["hi";"there";"you"];;
- : (int * string) list = [(2, "hi"); (3, "there"); (4, "you")]
# zipLists [2;3;4] [1;6;3;8];;
- : (int * int) list = [(2, 1); (3, 6); (4, 3)]
```

21. Write a function `mkassoc` which takes a key k, a value v, and a list $[k_0, v_0; k_1, v_1; \cdots; k_n, v_n]$ of key-value pairs, and returns a new list $[k, v; k_0, v_0; k_1, v_1; \cdots; k_n, v_n]$.

```
# let mkassoc k v xx = ...
val mkassoc : 'a -> 'b -> ('a * 'b) list -> ('a * 'b) list = <fun>
# let env = mkassoc "x" 3 (mkassoc "y" 4 []);;
val env : (string * int) list = [("x", 3); ("y", 4)]
```

22. Write a function `assoc` which takes a list of key-value pairs, a key, and returns the corresponding value if it exists in the list. You do not need to check if the key exists in the list.

```
# let rec assoc k xx = ...
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val assoc : 'a -> ('a * 'b) list -> 'b = <fun>
# assoc "x" env;;
- : int = 3
# assoc "y" env;;
- : int = 4
```

Note you will get a warning, since you did not handle the empty associative list. You can ignore this.

23. Write a function `substitute` which takes a list $[x_0; x_1; \cdots; x_n]$, an element x, and a replacement y, and returns a new list such that all elements equal to x are replaced with y.

```
# let rec substitute xx x y = ...
val substitute : 'a list -> 'a -> 'a -> 'a list = <fun>
# substitute [2;3;4] 2 9;;
- : int list = [9; 3; 4]
# substitute [2;3;5;2] 2 9;;
- : int list = [9; 3; 5; 9]
# substitute [2;3;5;2] 0 30;;
- : int list = [2; 3; 5; 2]
```

24. Write a function `interp` that interprets a list of elements. Your interpreter will have 3 commands. A non-negative integer will represent pushing the element onto the stack. A -1 will pop x and y off the stack, and push $x + y$ onto the stack. A -2 will pop x and y off the stack, and push $x - y$ onto the stack.

    Once the input list is empty, the interpreter should output the number at the top of its stack.

```
# let rec interp xx =
val interp : int list -> int = <fun>
# interp [ 10; 20; -1; 5; 2; -2; -1]
- : int = 33
```

3 Handing In

You will get an email about your account on `host220.cns.iit.edu`. Your file will need to be committed into the subversion repository hosted there. This information will be delievered on Monday.